Formal Methods: What's in it for me?

2005 FAA/NASA Software and Complex Electronic Hardware Conference

Research supported by NASA Langley Research Center and Honeywell Cooperative agreement NCC-1-399

Darren Cofer

Honeywell Laboratories
Minneapolis MN
darren.cofer@honeywell.com
612-951-7279

FM team
Samar Dajani-Brown
Eric Engstrom
Vu Ha
Murali Rangarajan

Honeywell

Outline

Honeywell

- Context
  - verification of safety-critical avionics software (and HW)
  - complex, real-time, fault tolerant
  - why do we need formal methods?
- Approach
  - model checking
  - tools
- Examples
  - real-time operating system for avionics
  - triplex sensor voter
- Conclusions
  - What's in it for me?

What's in it for me?

Honeywell

- Depends on who you are…
  - avionics manufacturer
  - airframer/integrator
  - airline
  - regulator
  - flying public
- …and what you do
  - engineer
  - QA/test
  - DER
  - management

- Attempt to answer from a viewpoint of a practitioner (rather than a researcher)
  - how to apply real tools to real systems for real airplanes

Formal methods

Honeywell

- **analysis is a necessary part of any engineering discipline**
- **formal methods = specification & analysis of SW designs**
- **why analyze software?**
  - **imagine building the mechanical components of an aircraft today without structural & aerodynamic analyses**
    - “We’ll just build it and see if it flies.”
  - **all software-related failures are due to design errors**
    - doesn’t break or wear out
    - testing and HW fault-models inadequate
  - **software is too easy to change**
    - susceptible to new errors at all life-cycle stages
  - **software errors are logical errors**
    - obscured by representation
    - difficult to detect errors by inspection

What is/are formal methods?

Honeywell

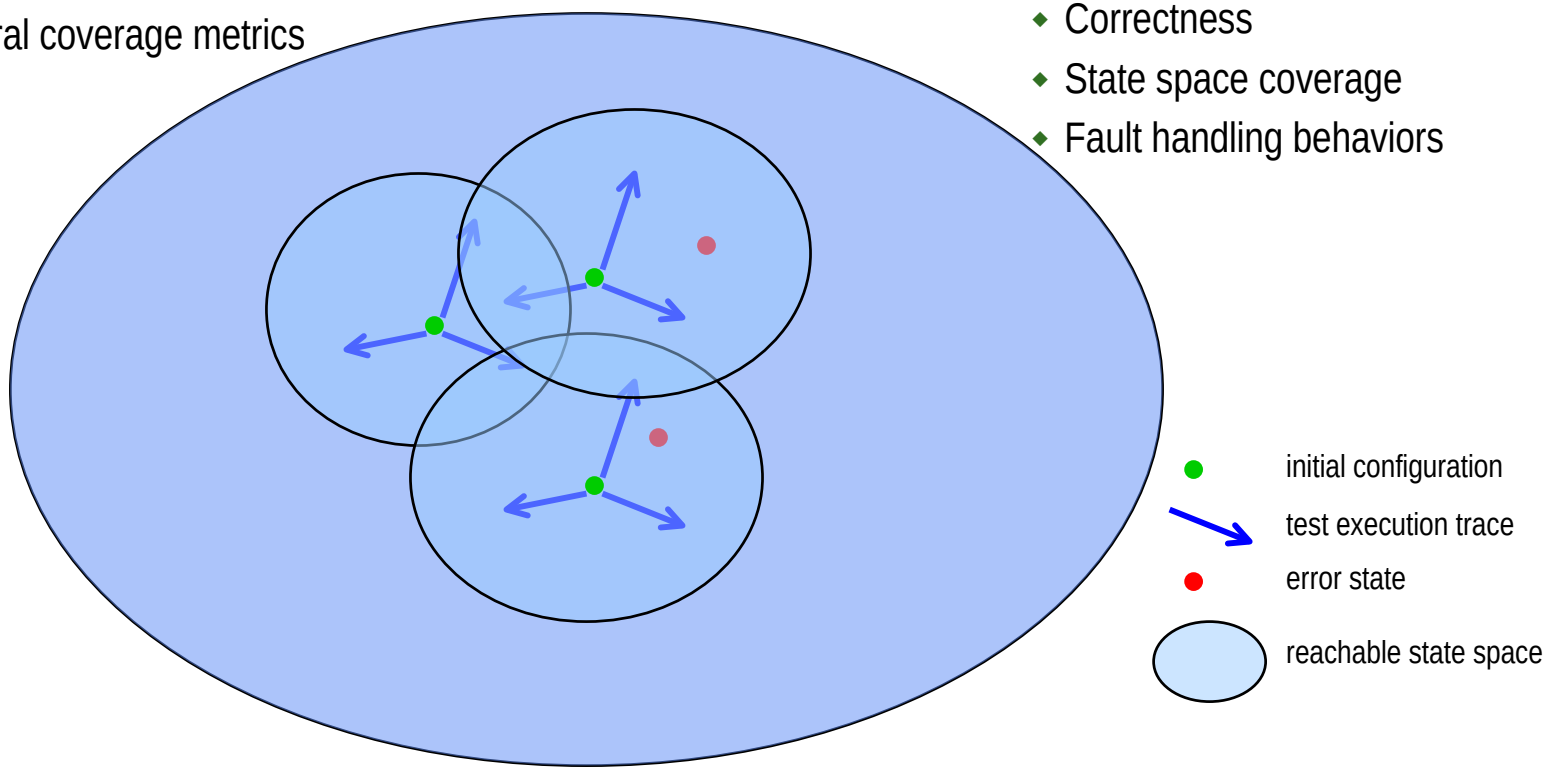
- using ideas and techniques from mathematics and formal logic
- to specify and reason about computing systems
- to increase design assurance and eliminate defects
- by allowing comprehensive analysis of requirements and design
- and complete exploration of system behavior
- including fault conditions
- augmented by automated SW tools

Test vs. formal analysis

Honeywell

- Inputs
  - Requirements + Test cases + Executable + Target platform
- Execute
- Outputs
  - Test results
  - Structural coverage metrics

- Inputs
  - Requirements + Design + Environment (assumptions)
- Reachability analysis
- Outputs
  - Correctness
  - State space coverage
  - Fault handling behaviors



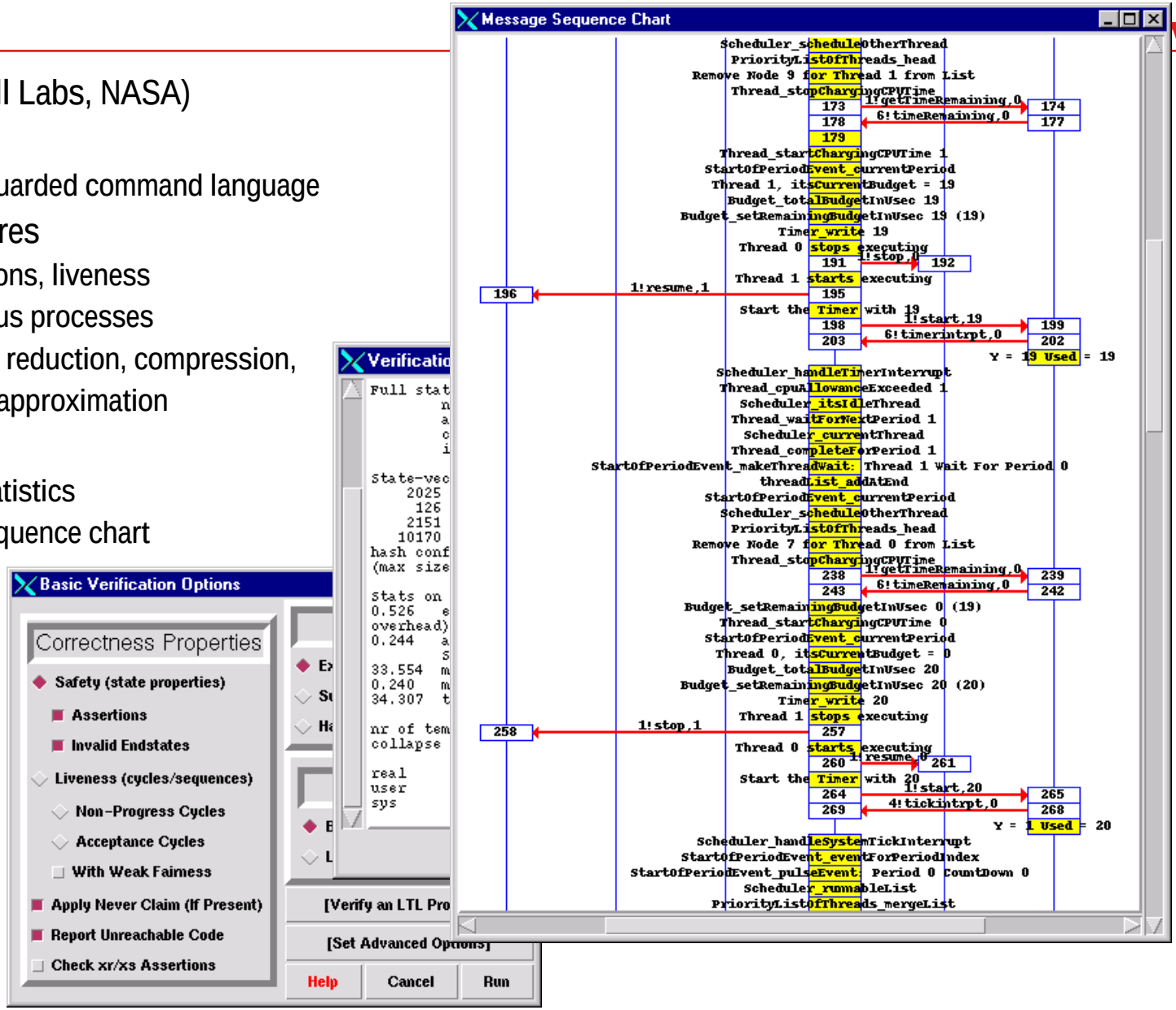
Model checking

Honeywell

- System modeled as automaton (state machine)
- Performs exhaustive state space search with various mechanisms for reducing state space
  - explicit state model checkers (Spin)
  - symbolic model checkers (SMV)
- Properties specified as
  - assertions on particular system states
  - path specifications (sequences of states – temporal logic)
- Checks if properties hold in any or all states or sequences of states
  - or that negation of property is not reachable in the model
- Originally developed for hardware and protocol verification
  - current research focusing on software verification
- Main advantage: produces counter-examples for aid in debugging
- Main disadvantage: state explosion problem
  - this problem is slowly being mitigated by various advanced techniques and Moore's law
- Approaches
  - model derived from design specification
  - model derived from implementation / code

Tools: SPIN

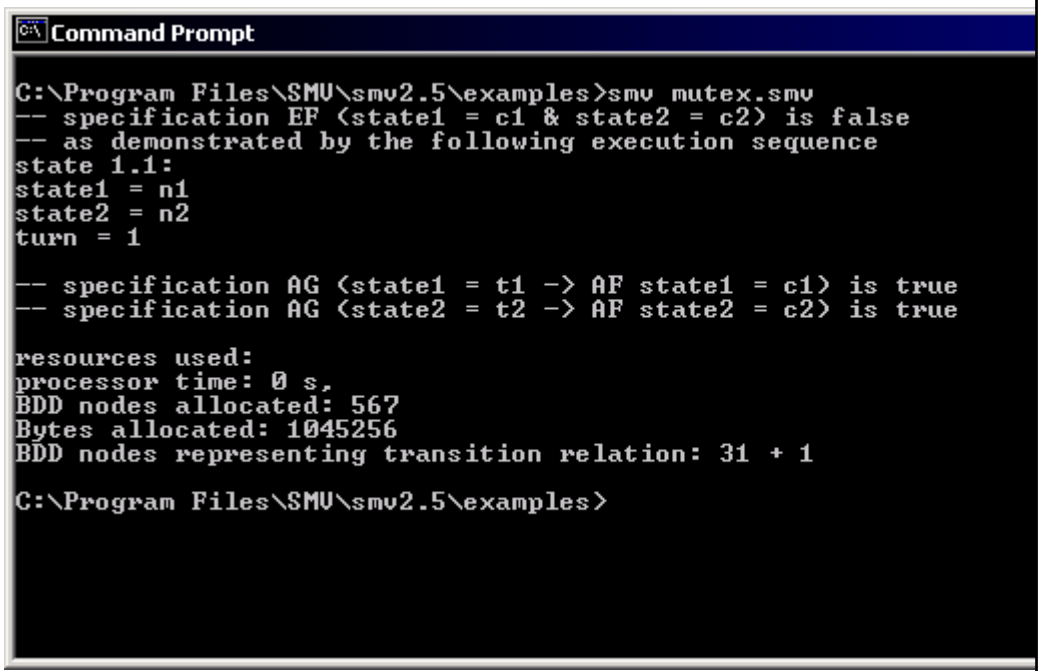
- Holzmann (Bell Labs, NASA)
- Input
  - Promela – guarded command language
- Analysis features
  - LTL, assertions, liveness
  - asynchronous processes
  - partial order reduction, compression, state space approximation
- Output
  - results & statistics
  - message sequence chart



Tools: SMV/NuSMV

Honeywell

- Clarke (CMU) and others
- Text interface
- synchronous data flow language
- CTL specs
- counter examples as sequence of states

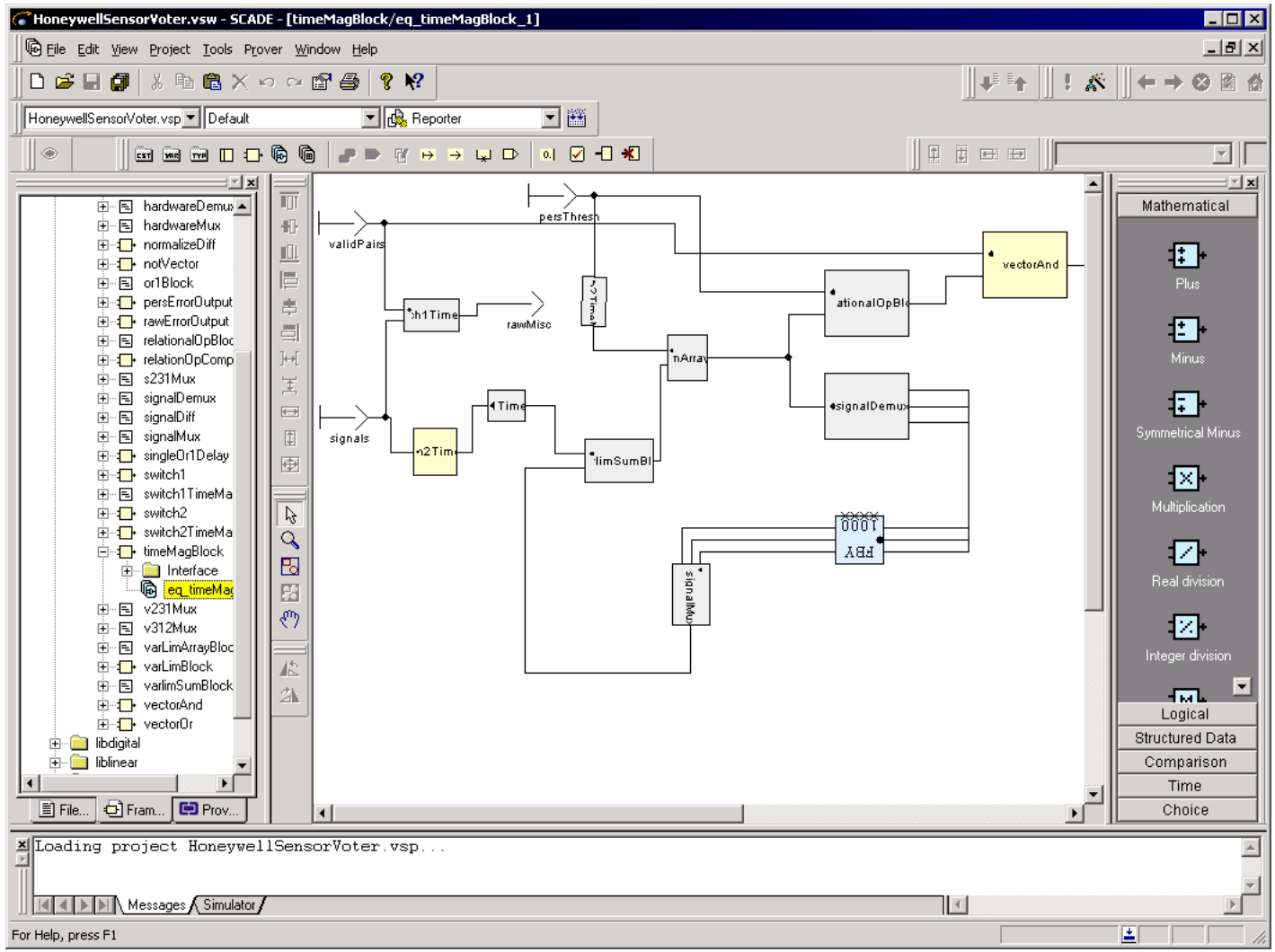



```
MODULE main

VAR
state1: {n1, t1, c1};

ASSIGN
init(state1) := n1;
next(state1) :=
case
   (state1 = n1) & (state2 = t2): t1;
   (state1 = n1) & (state2 = n2): t1;
   (state1 = n1) & (state2 = c2): t1;
   (state1 = t1) & (state2 = n2): c1;
   (state1 = t1) & (state2 = t2) & (turn = 1):  c1;
   (state1 = c1): n1;
   1: state1;
esac;

VAR
state2: {n2, t2, c2};

<…>

VAR
turn: {1, 2};

ASSIGN
init(turn) := 1;
next(turn) :=
case
   (state1 = n1) & (state2 = t2): 2;
   (state2 = n2) & (state1 = t1): 1;
   1: turn;
esac;

SPEC
EF((state1 = c1) & (state2 = c2))

SPEC
AG((state1 = t1) -> AF (state1 = c1))

SPEC
AG((state2 = t2) -> AF (state2 = c2))
```

Tools: SCADE

Honeywell

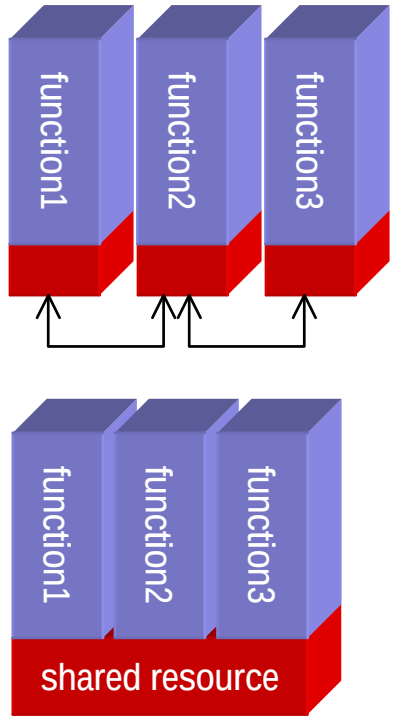
- Esterel Technologies
- Integrated CAD and verification
- graphical synchronous data flow language
- Design Verifier
- Import-export Simulink designs



Example 1: RTOS

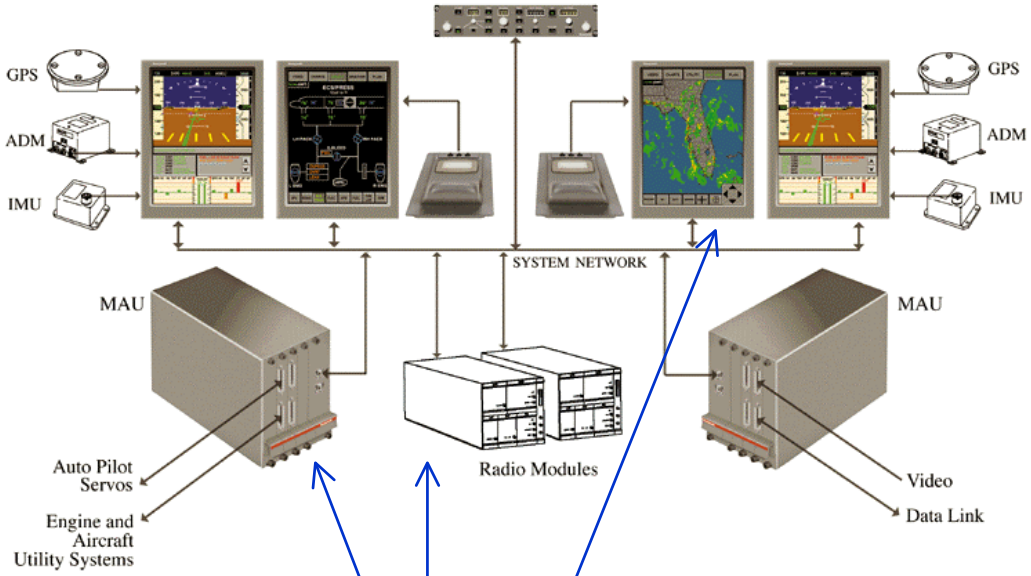
Honeywell

- Integrated Modular Avionics
  - applications of different criticalities share resources (CPU)
  - reduced cost, weight, maintenance (reduce amount of “level A” code)
- Real-time operating system with time partitioning
  - scheduler ensures that actions of one thread cannot impact other threads’ guaranteed access to resources
  - RT control tasks: every thread gets its allotted CPU budget every period
  - benefits: fault containment, system upgrades
- Why use formal methods
  - concurrency, complexity, real-time guarantees
  - high integrity required (level A)
- Goal
  - demonstrate effectiveness of model checking to analyze key properties of safety-critical software that would be difficult or impossible to establish by traditional means
  - advanced debugging: augments existing development process



Primus Epic integrated avionics system

Honeywell



Business, regional, and commuter aircraft

- *Bombardier Global Express*
- *Raytheon Hawker Horizon 450*
- *Agusta-Bell AB-139*
- *Gulfstream V-SP*
- *Sino Swearingen SJ30-2*
- *Cessna Citation Sovereign*

- Deos real-time OS
  - Primarily business, regional, commuter aircraft
  - Primary flight controls, autopilot, displays, navigation...
  - Rate monotonic scheduling with priority inheritance
  - Time & space partitioning, dynamic threads, slack scheduling, aperiodic interrupts, mutexes
  - Portable to various COTS CPUs
  - DO-178B Level A

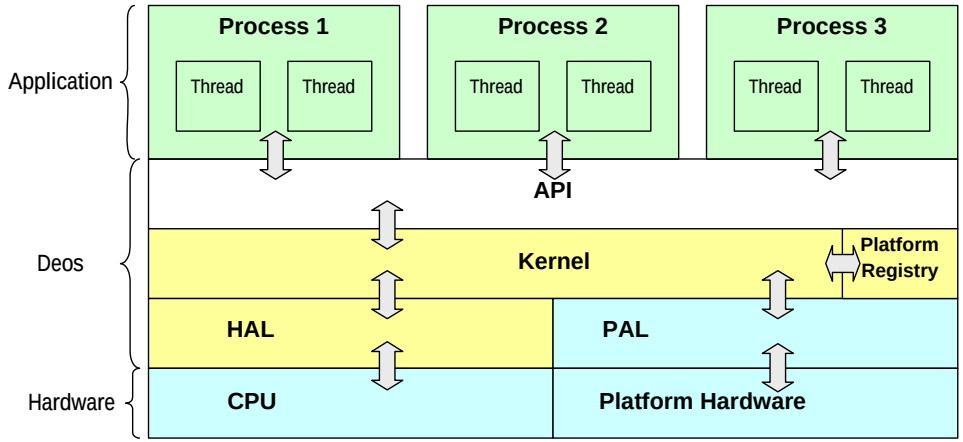
Unique aspects of this work

Honeywell

- **Software analyzed is an actual IMA RTOS**
  - ***Deos* – currently in service in several aircraft product lines**
- **Model derived directly from C++ source code**
  - **analysis results closely linked to real system**
- **Overhead accounting computations explicitly modeled**
  - **critical to timing guarantees**
- **Advanced scheduling features**
  - **complicates both implementation and analysis**

Deos components and terminology

Honeywell



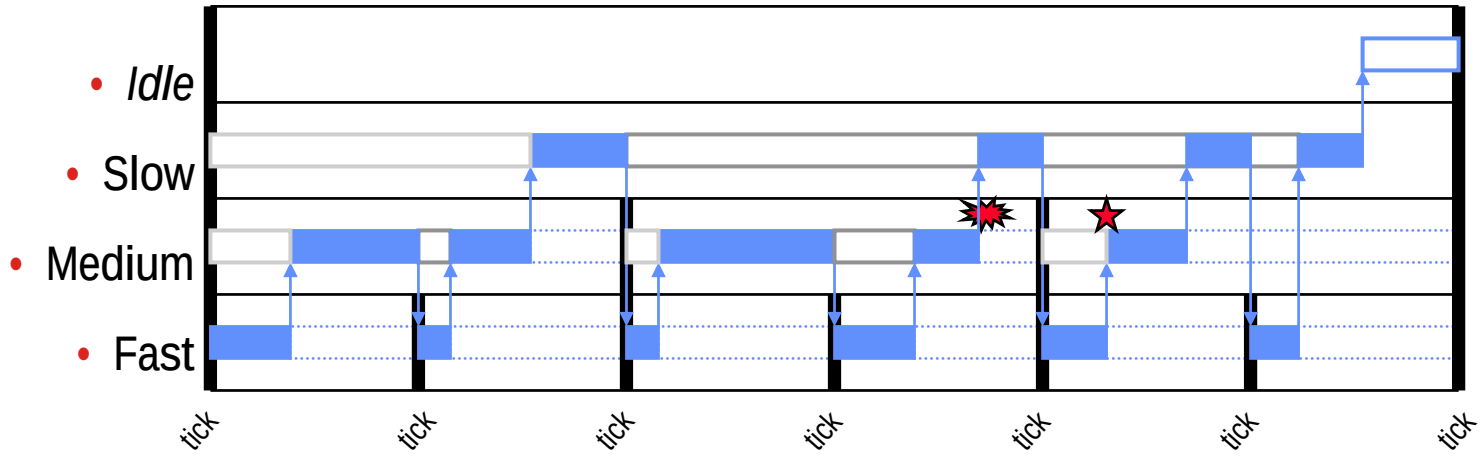
- Process – basic entity for resource mgt. (time, memory, mutexes, I/O,…)
- Thread – sequential execution unit (function + period + time budget)
- API – application programmer's interface, governs & validates interactions with OS
- HAL – HW abstraction layer, interface to CPU
- PAL – platform abstraction layer, interface to timers and I/O
- Registry – system configuration parameters
- Kernel – scheduler, memory management, etc.

Everything you need to know about...

- ***Rate Monotonic Scheduling***
  - **Always run the highest priority thread that is ready to run**
  - **All threads are periodic and are guaranteed to receive a statically specified CPU budget *every period***
    - specify the worst case execution time (WCET) for safety critical threads
  - **Programmers do not assign priorities to threads**
    - thread priority is inferred from period
    - threads with shorter periods have higher priorities
  - **Thread notifies kernel when it has finished its execution for a period**
    - kernel preempts a thread that attempts to execute beyond its CPU budget
  - **Deos uses harmonic periods**
    - permits ~100% utilization of CPU time

RMS – pictorially

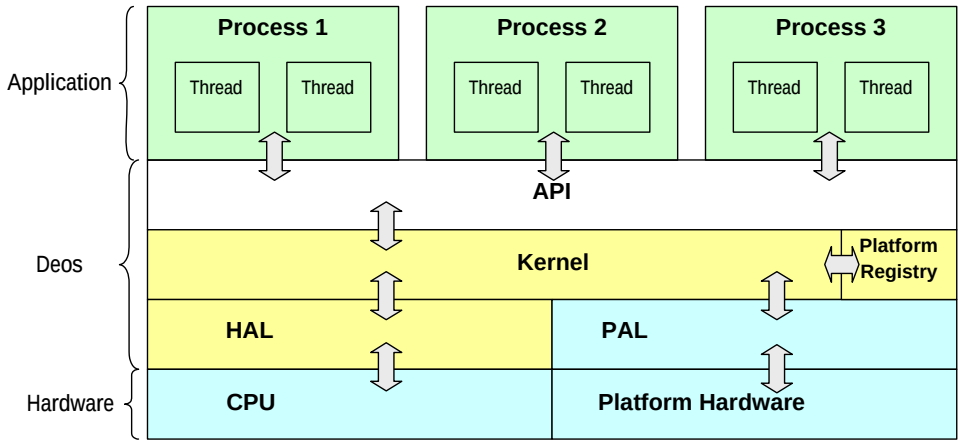
Honeywell



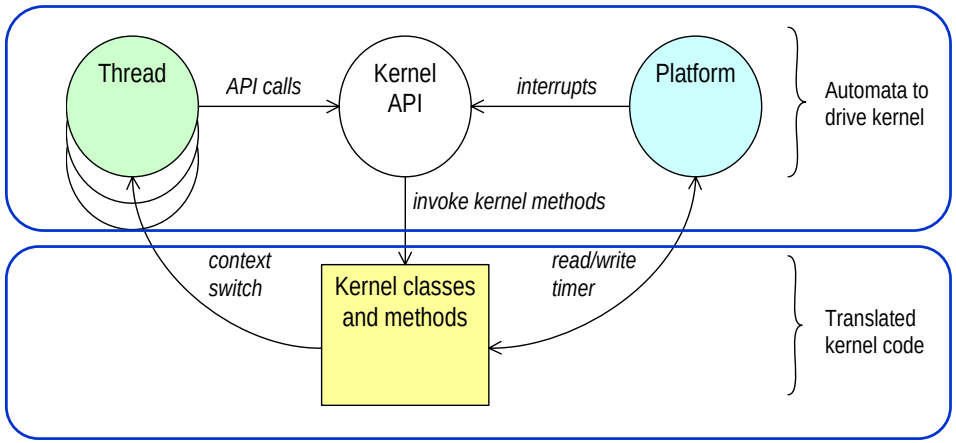
- Thread is executing
- Thread is ready for execution but has not yet been activated by scheduler
- Thread has been preempted by higher priority thread
- Thread has completed for period
- Thread completes for period, activate another thread
- Thread preempted by higher priority thread
- Thread consumes budget without completing for period (TBE)
- Thread resumed with propagation of TBE exception

Structure of model

Honeywell



Real system



Model

- **Separation of system and environment**
- **System modeled with high fidelity, traceability to real system**
  - certification
  - implementation, autocoding
- **Environment**
  - abstract, only model necessary parts
  - fault injection

Environment

System to be verified

High fidelity model of software

Honeywell

- **Model derived from kernel code (rather than specification)**
  - **Accuracy (design = code)**
  - **Utility in verification/certification**
  - **Level of detail required to capture subtle timing behaviors**

```
void Scheduler::initializeInterruptEvents()
{
  interruptEvents =
   new InterruptEvent*[ numInterrupts ];

  for ( platformInterruptNumber i = 0;
                 i < numInterrupts; i++ )
  {
    interruptEvents[i] = new InterruptEvent(
i );
    ignorePlatformInterrupt[i] = false;
  }
}
```

Deos scheduler
- 55 files
- 800KB C++ files
- ~15K LOC

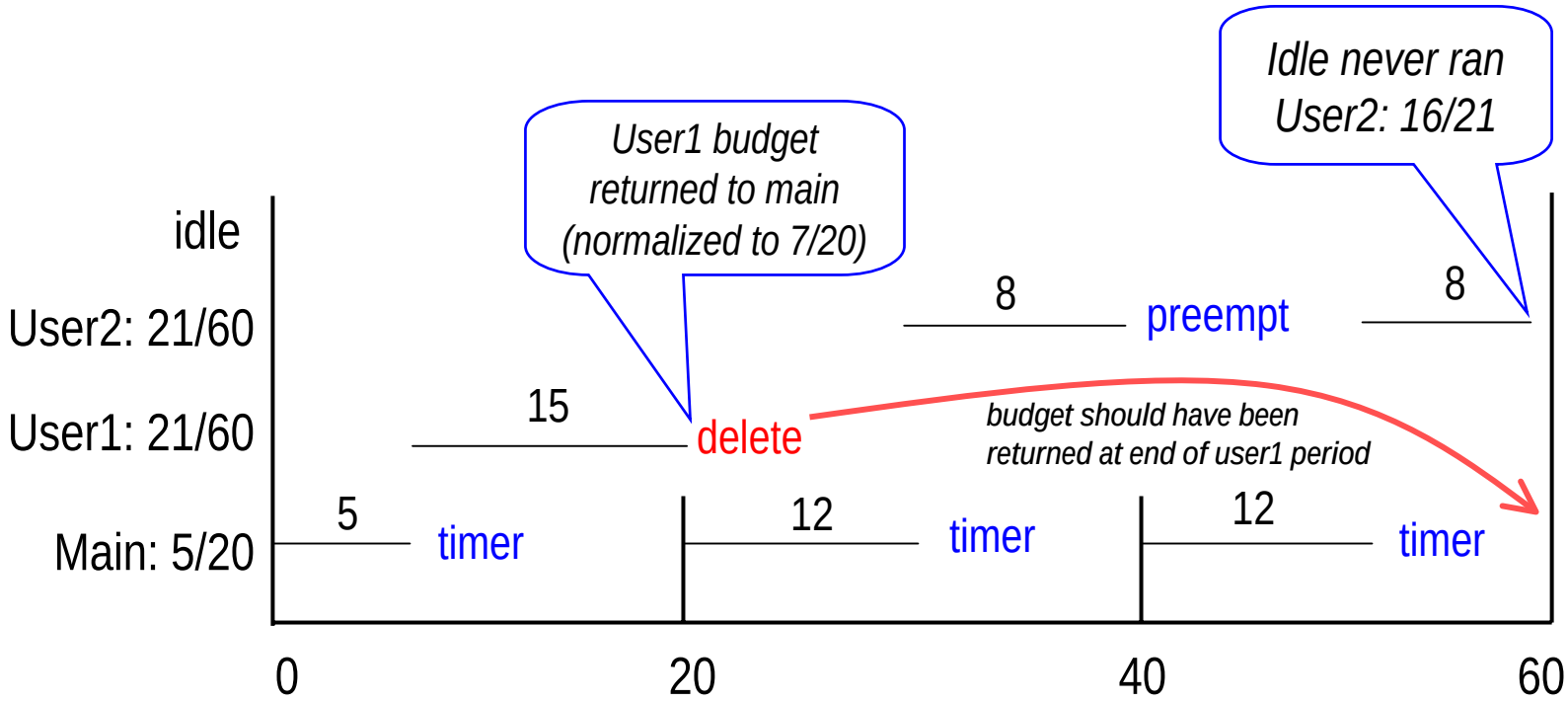
```
inline Scheduler_initializeInterruptEvents()
{
  InterruptEvent_PTR
   Scheduler_interruptEvents[InterruptEvent_MAX];

  byte i = 0;
  do
  :: (i < InterruptEvent_MAX) ->

NEW_InterruptEvent(Scheduler_interruptEvents[i]);
    i++;
  :: else -> break;
  od;
}
```

SPIN model
- 20 classes
- 120 methods
- >4000 LOC (Promela)

```
inline Thread_waitForNextInterrupt() {
assert(currentBudget.remainingBudget >= 2*contextSwitchPlusDelta+cacheBonus);
```

Time partitioning error trace

Honeywell




```
LTL property -- time partitioning
[](startPeriod -> (!endPeriod U idleRun))
```

Penix, Visser, Engstrom, Larson, Weininger, "Verification of Time Partitioning in the DEOS Scheduler Kernel," ICSE 2000

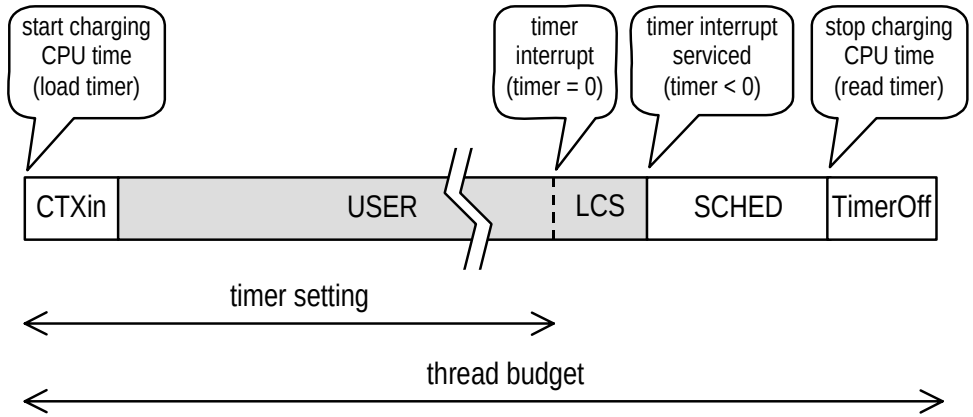
Time partitioning

Honeywell

- **With slack and ISRs, idle thread may never run: need new approach**
  - **Assertion checked in the tick interrupt handler**
  - **Detects the problem after scheduler has over-committed**
  - **Consists of 2 parts –**
    - one for currently running thread
    - another for threads in all periods ending at this tick
  - **Disjunction of 3 conditions –**
    - thread is the Idle thread (no deadline)
    - thread received its full budget (remaining budget = 0)
    - thread voluntarily completed for period
- **No errors found**
  - **Doesn't hold for ISR threads**
    - can't guarantee interrupt occurs early enough in period
    - but interrupts don't interfere with other threads

Overhead accounting

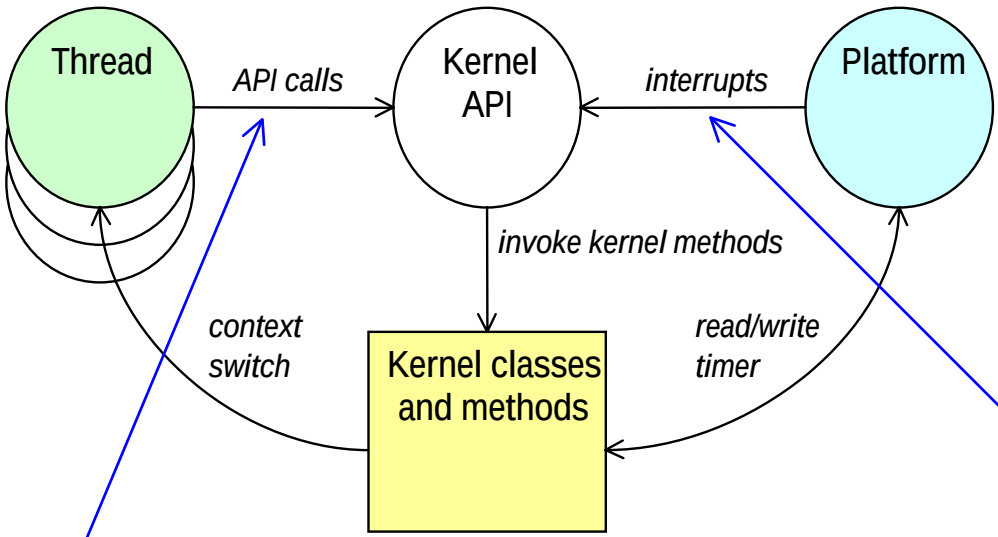
Honeywell



- **Scheduler must ensure that time spent executing a thread (including kernel overhead) does not exceed thread budget**
  - **Timer loaded when thread is dispatched, counts down to 0**
    - response delayed by critical section, e.g. kernel service (LCS)
    - scheduling actions to pick next thread (SCHED)
    - read timer, decrement thread budget, load timer for next thread (TimerOff)
  - **Pre-deduct overhead quantities from thread budget at dispatch ("contextSwitchPlusDelta")**
    - minimum time required to execute a thread that does nothing (USER = 0)
  - **Implies minimum budget a thread must have to be placed on runnable queue**
    - ISR thread must have 2x this amount since it compensates interrupted thread for both context switches
    - more complicated for slack-consuming threads

Scheduler operations & overhead

Honeywell



⇩ Traps (SW interrupts)

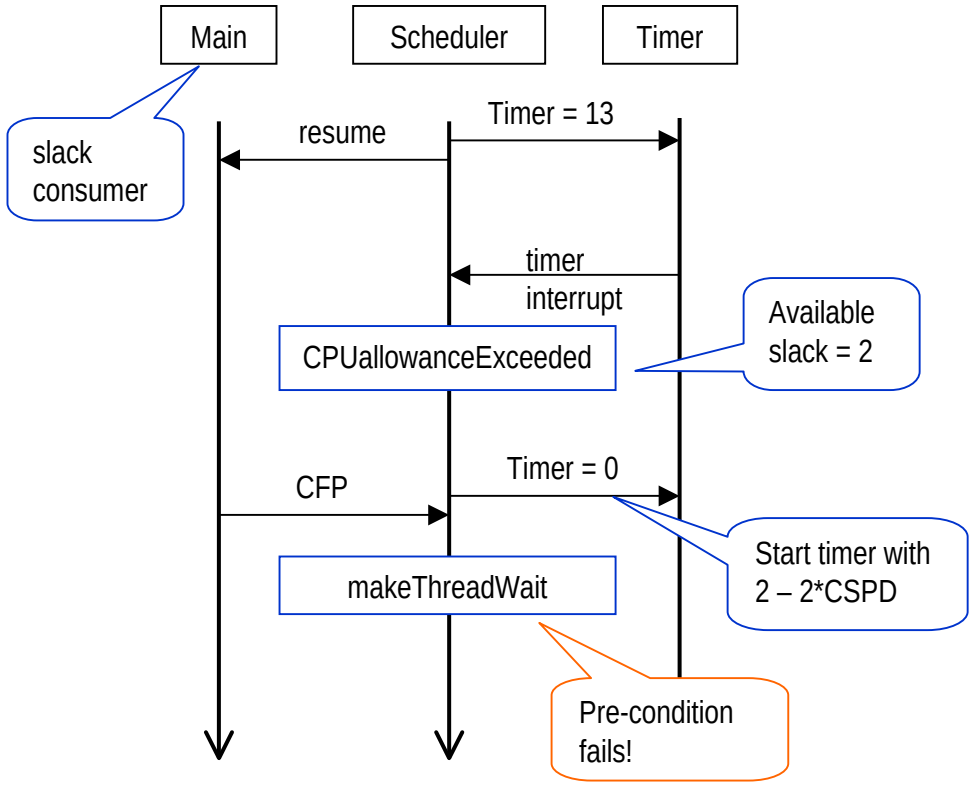
- Triggered when user thread attempts to execute kernel service
- Services may include critical sections (interrupts disabled)
- charge time to calling thread

- **System tick interrupt**
  - Generated periodically by platform hardware
  - Tick handler may cause currently running thread to be preempted
  - compensate running thread from total CPU utilization
- **Timer interrupt**
  - Generated by platform hardware
  - Produced when thread timer runs down to zero
  - charge to thread
- **User interrupts**
  - Generated asynchronously by I/O or other hardware
  - May cause currently running thread to be preempted by ISR thread
  - ISR compensates interrupted thread

Precondition for SOPEvent::makeThreadWait()

Honeywell

- **makeThreadWait() precondition:**
  - remaining budget ≥ contextSwitchPlusDelta
  - **available slack = 2**
  - **main thread uses budget and requests slack**
  - **2*contestSwitchPlusDelta pre-deducted from available slack, leaving 0**
  - **main decides to immediately complete for period**
  - **when adding main to startOfPeriodEvent queue the precondition is violated**



Action:

• when makeThreadWait() is called, the time remaining for the thread has not yet been compensated to add contextSwitchPlusDelta

• precondition changed to: remainingBudget ≥ 0

Example 1: summary

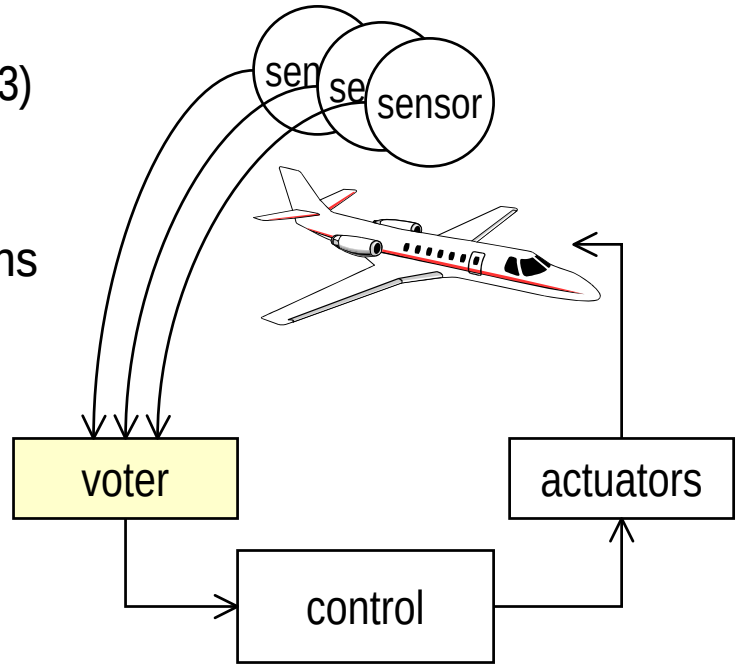
Honeywell

- **Verified the main time partitioning assertion and function preconditions for many different system configurations, including interrupt threads and slack scheduling**
  - Increased assurance that this complex system has been designed and implemented correctly
- **Identified several instances where preconditions were inconsistent with the intended operation of the scheduler**
  - These have been corrected and will improve the quality of code reviews performed in future verification and certification activities
- **Identified a number of modeling errors**
  - Enabled us to refine the model to reflect realistic system behaviors and increased confidence in our analysis results
- **Detected several unexpected system behaviors**
  - Improves our understanding of the operation of the system – helpful for maintenance and system upgrades

Example 2: Triplex sensor voter

Honeywell

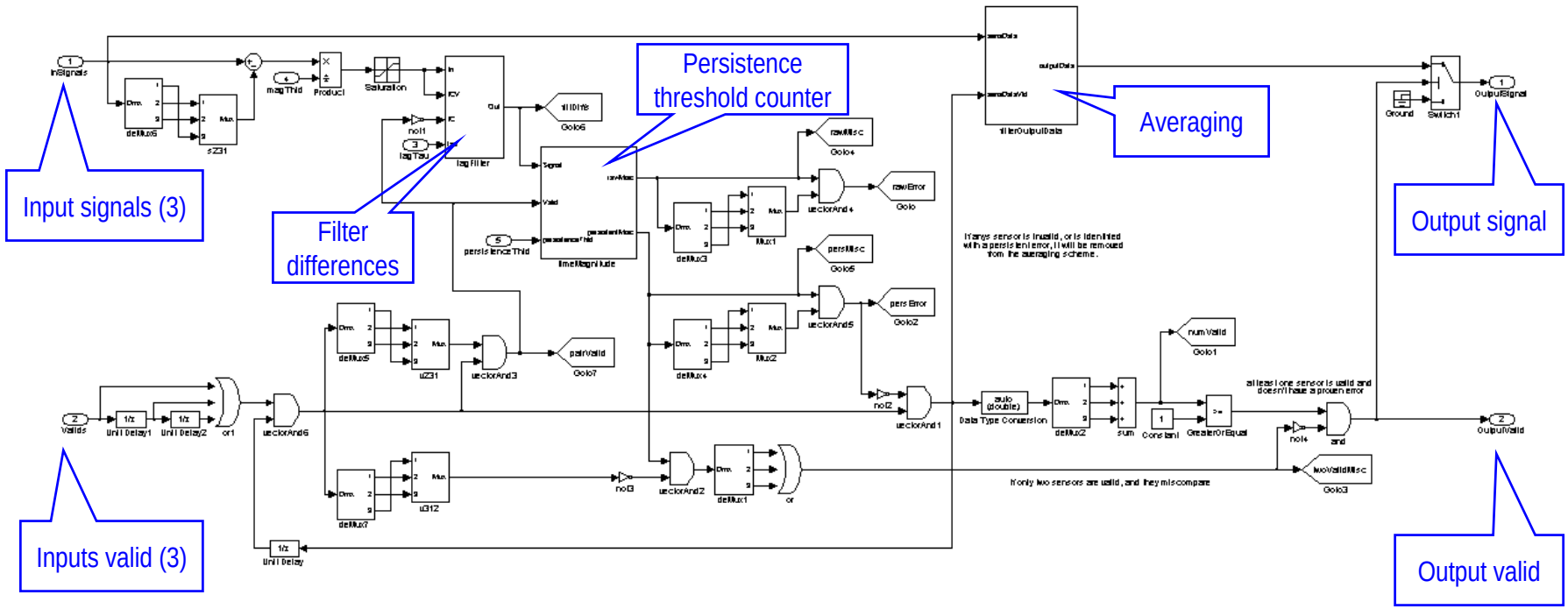
- Problem
  - Algorithm for management of redundant sensors (3)
    - air data, inertial reference, etc.
  - Design captured in Simulink
  - Mixture of algorithmic features from several designs
- Why use formal methods
  - Fault handling behavior
  - High integrity required (level A)
  - Leverage Model-Based Development:
    - Flight SW can be autocoded from design
- Objectives
  - Comprehensive assessment of design correctness
  - Assess automation of translation/verification process
  - Tool evaluation (SMV, SAL, SCADE)



Sensor voter operation

Honeywell

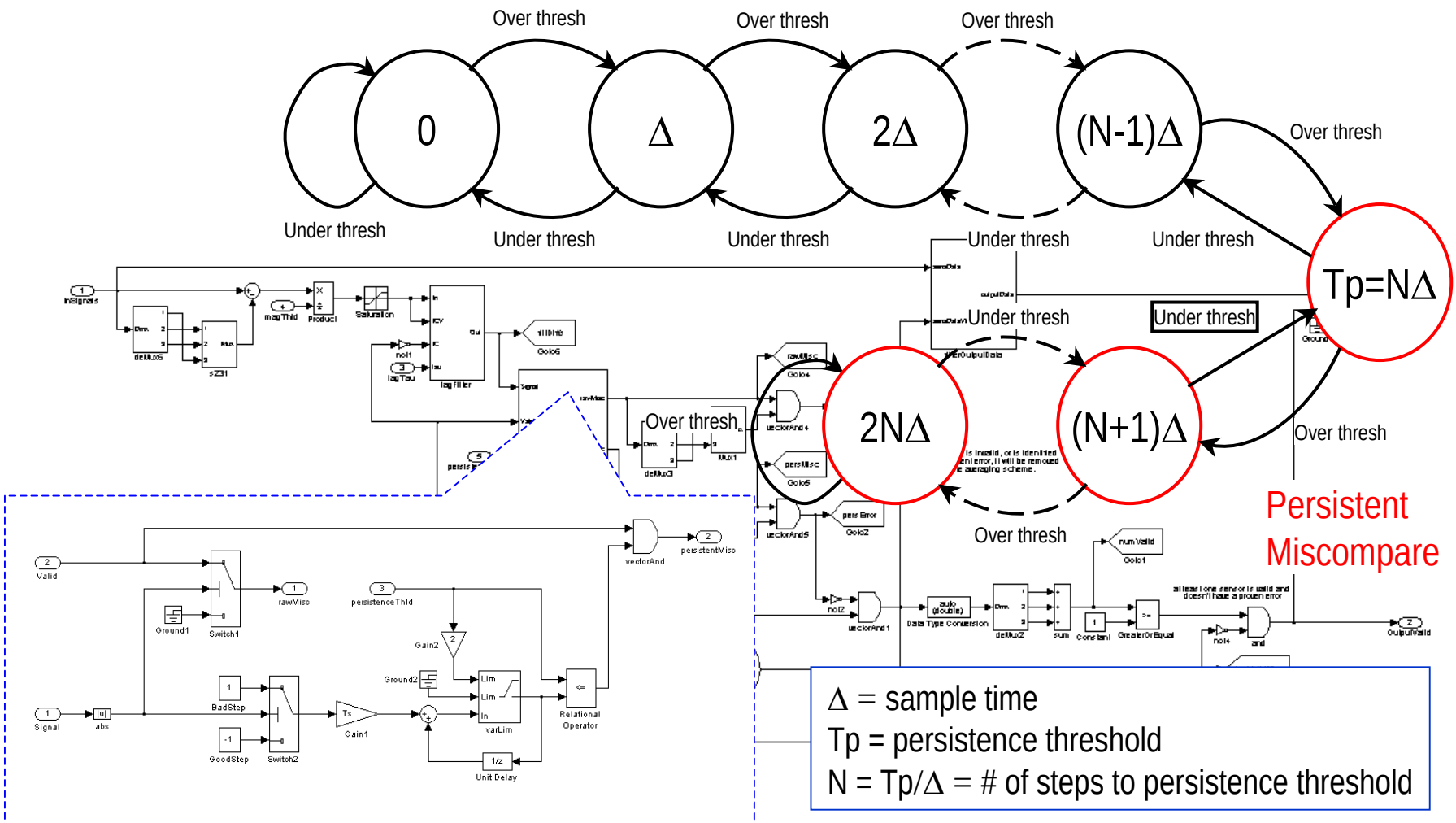
- Digitized sensor signals sampled at 20 Hz
- Valid flag supplied by sensor hardware
- Use valid flag and comparison of redundant measurements to detect and isolate sensor faults
- Tolerate “false alarms” due to noise, transients, small differences
- Output – composite robust sensor output and valid flag



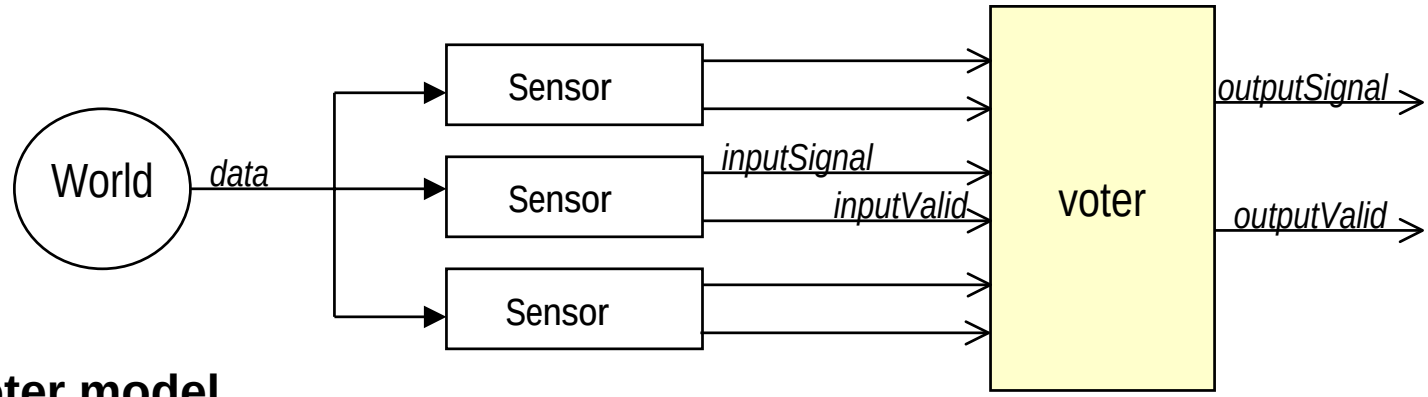
Persistence threshold counter

Honeywell

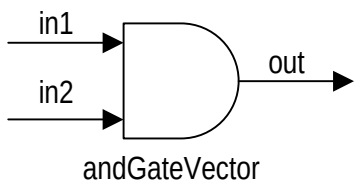
- **Find the "hidden" state machine...**



Voter model



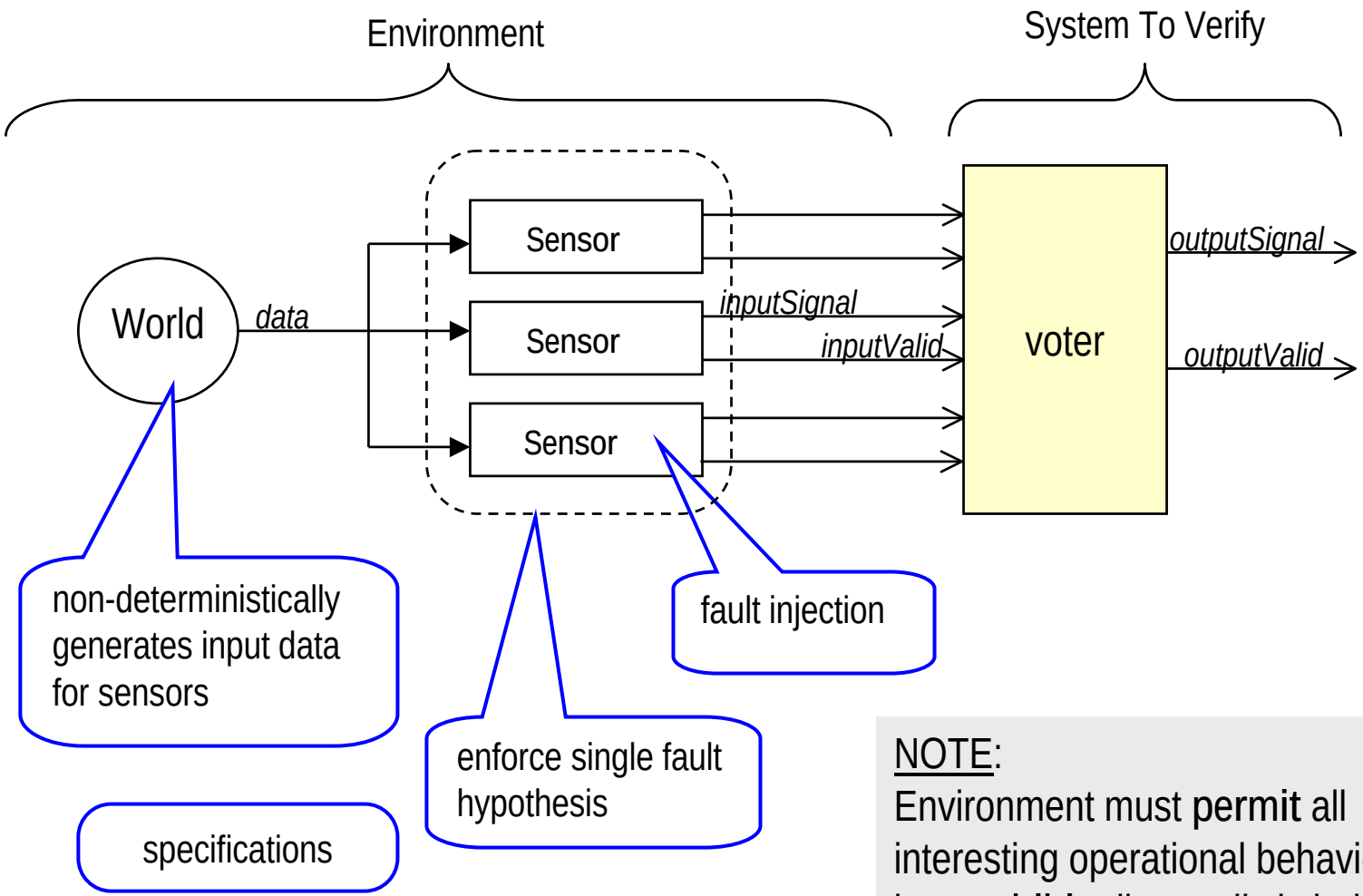
- **Voter model**
  - capture behavior of sensor voter algorithm
- **Sensor model (environment)**
  - measure 'true' world data and provide signals to voter
  - fault injection – valid flags, noise, signal errors
- **Output signal requirements**
  - synthesize correct output signal
  - limit transients in output signal
- **Fault handling requirements**
  - detect sensor faults and isolate faulty sensor
  - must not produce false alarms due to transient conditions




```
MODULE andGateVector(in1, in2)
VAR
  out : array 0 .. 2 of boolean;
ASSIGN
 out[0] := in1[0] & in2[0];
 out[1] := in1[1] & in2[1];
 out[2] := in1[2] & in2[2];
```

Environment model

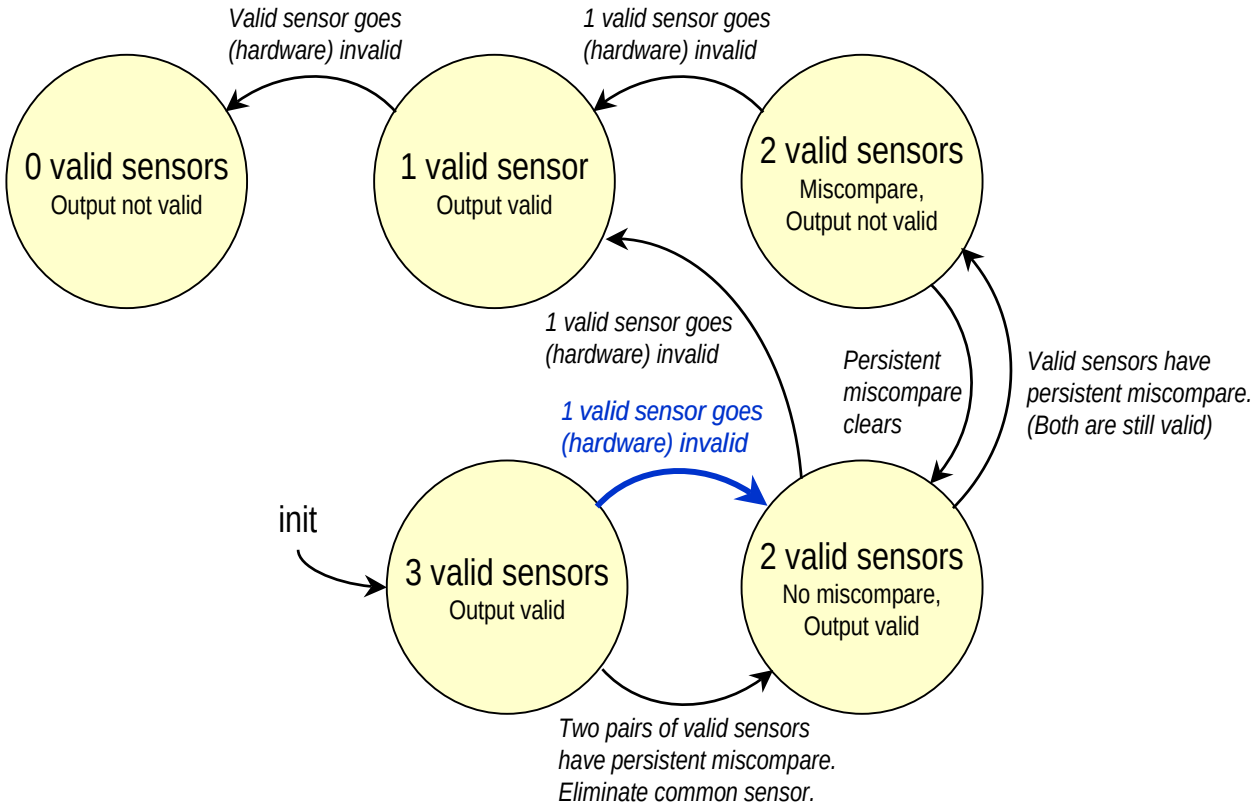
Honeywell



NOTE:
Environment must **permit** all interesting operational behaviors, but **prohibit** all unrealistic behaviors

Fault-handling requirements

Honeywell

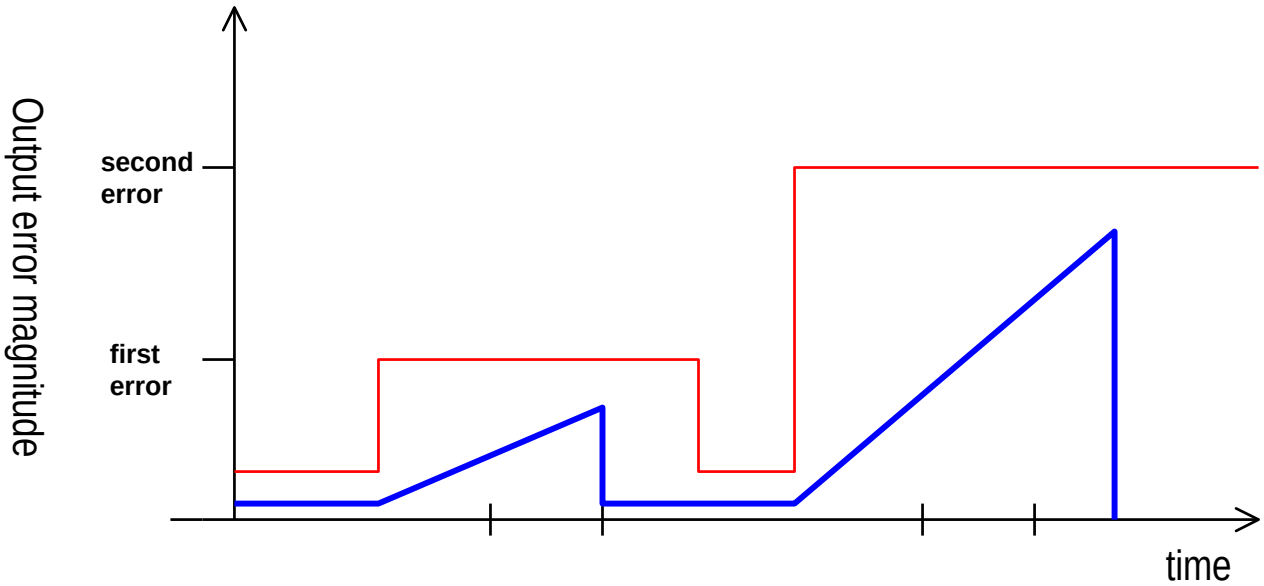



```
SPEC
AG((sensorVoter.numValid=3)^(~sensor1.valid)->
   AF(sensorVoter.numValid=2))
```

Computational requirements

Honeywell

- **Does computed output agree with the true environmental input data?**
  - **within some signal tolerance**
  - **within some time bounds for transients**
- **Transient may occur at time of sensor faults**
  - **requirements more strict for first fault**



Example 2: summary

Honeywell

- Precise definition of voter requirements
- Verified computational requirements
- Verified fault handling requirements
- Assessment of 3 model checking tools

Back to “What’s in it for me?”

Honeywell

- **Why should we support use of formal methods?**
  - **software engineering best practices**
    - strengthens “analysis” activities, quantitative assessment of design
    - get it right the first time
  - **current processes/guidelines not adequate for complex systems**
  - **industry will use if there is a cost advantage**
    - reduced design errors
    - early detection of errors
    - improve process predictability
    - facilitates reuse
    - automation of manual reviews
    - alternative to unit test
  - **certification authorities should be prepared**
    - proposed changes in DO-178C

DO-178B guidelines

- *Quality control* – identify and eliminate design faults
  Formal analyses and proofs = “other means” allowed in Section 6.2 to satisfy verification process objectives in the case of *complex behaviors that are not amenable to testing*
- *Quality assurance* – identify implementation errors
  Use formal techniques and tools to perform the *reviews and analyses* of Section 6.3.2, identifying and verifying requirements for functions and components as well as assumptions regarding their environment

Formal methods forecast

Honeywell

- **Short term – What can we do now?**
  - **practical engineering tool**
    - advanced debugging tools for early defect detection/elimination
  - **augment current processes to increase assurance**
  - **early design verification**
    - provides a framework for accurate implementation
  - **some domain specific tools and automation**
    - SCADE, extensions to Simulink
- **Long term – Where do we want to be?**
  - **model-based development**
  - **fully integrated into development process**
  - **invisible?**
  - **certification credit for formal analysis**
    - alternative to unit testing in some cases
    - analysis of design, distinct from implementation

Honeywell

www.honeywell.com